Jacobson, Emerson
Santini, Nicholas
CS 202

Lab 1 Report

Video

https://youtu.be/B8JlzyTJXQI

Changes

The following is a list of files that have been changed, along with what was changed in each file.

Makefile

```
        $U/_grind\
        $U/_wc\
        $U/_zombie\
+       $U/_lab1_test\

fs.img: mkfs/mkfs README $(UPROGS)
        mkfs/mkfs fs.img README $(UPROGS)
```

The only thing that was changed in this file was the addition of the lab1_test user program.

kernel/defs.h

```
  struct sleeplock;
  struct stat;
  struct superblock;
+ struct pinfo;

  // bio.c
  void            binit(void);

  void*           kalloc(void);
  void            kfree(void *);
  void            kinit(void);
+ int             get_free(void);

  // log.c
  void            initlog(int, struct superblock*);

  int             either_copyout(int user_dst, uint64 dst, void *src, uint64 len);
  int             either_copyin(void *dst, int user_src, uint64 src, uint64 len);
  void            procdump(void);
+ void            add_call(int);
+ int             sysinfo(int);
+ int             procinfo(struct pinfo*);

  // swtch.S
  void            swtch(struct context*, struct context*);
```

New definitions of functions that were created throughout the lab were added to this file. Specifically: pinfo - for the procinfo syscall, get_free - a helper function to get the number of free pages, add_call - another helper function to count the number of syscalls, sysinfo - the first syscall we were to implement, and procinfo - the second syscall we implemented.

kernel/kalloc.c

```
      memset((char*)r, 5, PGSIZE); // fill with junk
    return (void*)r;
  }
+
+ int get_free(void) {
+   struct run *r = kmem.freelist;
+   if (r == 0) return 0; // Return if null
+
+   int num = 0;
+
+   acquire(&kmem.lock);
+   while (r != 0) {
+     num++;
+     r = r->next;
+   }
+   release(&kmem.lock);
+
+   return num;
+ }
```

This is where the get_free() helper function was implemented that counts the number of free memory pages. It starts with the address of kmem.freelist and iterated over it until it reaches a null pointer. It takes the kmem lock as well to avoid any possible conflicts during execution.

kernel/proc.c

```
        initlock(&p->lock, "proc");
        p->state = UNUSED;
        p->kstack = KSTACK((int) (p - proc));
+       p->syscalls = 0;
    }
  }

```

```
      printf("\n");
    }
  }
+
+ static int num_calls = 0;
+
+ void add_call(int num) {
+   num_calls++;
+ }
+
+ int sysinfo(int in) {
+   if (in == 0) {
+     int count = 0;
+     struct proc *p;
+
+     for(p = proc; p < &proc[NPROC]; p++) {
+       if (proc->state != UNUSED) count++;
+     }
+     return count;
+   } else if (in == 1) {
+     return num_calls;
+   } else if (in == 2) {
+     return get_free();
+   }
+   return -1;
+ }
```

```
+ int procinfo(struct pinfo *in) {
+   if (in == 0) return -1;
+
+   struct proc *p = myproc();
+
+   if (p == 0) return -1;
+
+   struct pinfo data;
+
+   int pg_count = p->sz >> PGSHIFT; // pg_count = p->sz / PGSIZE
+   if (p->sz % PGSIZE != 0) pg_count++; // Account for loss of decimal
+
+   data.ppid = p->pid;
+   data.page_usage = pg_count;
+   data.syscall_count = p->syscalls;
+
+   return copyout(p->pagetable, (uint64)in, (char *)&data, sizeof(struct pinfo));
+ }
```

This is where the syscalls, sysinfo and procinfo, were implemented. Additionally the helper function add_call was implemented here. The number of syscalls that a process has made is also initialized to 0.

For the sysinfo call, and depending on the value of the argument passed to it, it will either: a) Count the used processes, b) return the number of syscalls that have been made by using the num_calls variable that is incremented whenever a syscall is made, or c) return the number of free memory pages by using the get_free helper function.

kernel/proc.h

```
    struct file *ofile[NOFILE];  // Open files
    struct inode *cwd;           // Current directory
    char name[16];               // Process name (debugging)
+   uint64 syscalls;
+ };
+
+ struct pinfo {
+   int ppid;
+   int syscall_count;
+   int page_usage;
  };
```

In this header file, the number of syscalls was added to the PCB struct, and the pinfo struct was defined. The syscalls element is added and used to keep track of the number of syscalls that a process has made.

kernel/syscall.c

```
  extern uint64 sys_link(void);
  extern uint64 sys_mkdir(void);
  extern uint64 sys_close(void);
+ extern uint64 sys_sysinfo(void);
+ extern uint64 sys_procinfo(void);

  // An array mapping syscall numbers from syscall.h
  // to the function that handles the system call.

  [SYS_link]    sys_link,
  [SYS_mkdir]   sys_mkdir,
  [SYS_close]   sys_close,
+ [SYS_sysinfo] sys_sysinfo,
+ [SYS_procinfo] sys_procinfo,
  };

  void

      // Use num to lookup the system call function for num, call it,
      // and store its return value in p->trapframe->a0
      p->trapframe->a0 = syscalls[num]();
+
+     // Increment syscall count AFTER doing the syscall
+     // to avoid counting the current call.
+     add_call(num);
+     p->syscalls++;
    } else {
      printf("%d %s: unknown sys call %d\n",
              p->pid, p->name, num);
```

In this file, the accounting of syscalls was added (to the syscall function) as well as the 2 new syscalls added to the array of syscalls. The add_call function will increment the total number of syscalls that have been made, and p->syscalls++ uses the syscalls element of the process to keep track of syscalls on a per-process basis.

kernel/syscall.h

```diff
 #define SYS_link    19
 #define SYS_mkdir   20
 #define SYS_close   21
+#define SYS_sysinfo  22
+#define SYS_procinfo 23
```

In this file we added 2 new entry numbers for the 2 new syscalls that we created.

kernel/sysproc.c

```diff
   release(&tickslock);
   return xticks;
 }
+
+uint64 sys_sysinfo(void) {
+  int in;
+
+  argint(0, &in);
+  return sysinfo(in);
+}
+
+uint64 sys_procinfo(void) {
+  uint64 in;
+  argaddr(0, &in);
+  procinfo((void *)in);
+  return 0;
+}
```

In this file we added the “glue” functions that process the arguments that collect the call arguments and call the syscall function themselves.

`user/user.h`

```
  struct stat;
+ struct pinfo;

  // system calls
  int fork(void);

  char* sbrk(int);
  int sleep(int);
  int uptime(void);
+ int sysinfo(int);
+ int procinfo(struct pinfo*);

  // ulib.c
  int stat(const char*, struct stat*);
```

In this file the 2 new syscalls so they are usable in user-mode programs.

`user/usys.pl`

```
  entry("sbrk");
  entry("sleep");
  entry("uptime");
+ entry("sysinfo");
+ entry("procinfo");
```

In this file 2 new lines were added to account for the 2 new syscalls that were added.

Note

We *technically* omitted the `lab1_test` file. Because we did not modify this file, and because it is the same for every group, we did not find it important enough to include.

Syscall Processes

User-Mode to Kernel

We first start at the user-level with a call to the `sysinfo` syscall. Using the definitions in `user/usys.pl` and `user/user.h`, the correct syscall index is found (via `kernel/syscall.h`) This index is then used in the `syscall` function (defined in kernel/syscall.c) to index into the syscalls array (a "mapping" of syscall numbers to their respective functions). Before we run the system call itself, we first need to collect the arguments to pass to it. This is done in their respective sys_ functions; sys_sysinfo and sys_procinfo for the 2 syscalls that we implemented. After the correct arguments have been collected, they are passed to the syscalls themselves. In the next 2 sections we discuss the specific implementations of the system calls that we implemented that happen before the return to user-mode.

sysinfo

For the sysinfo we have 4 different return values depending on the incoming argument. If in is 0 we count the number of active processes. We consider a running process to be one that does not have the state of UNUSED in the list of process blocks. If in is 1, then we return the value of num_calls. num_calls is incremented every time that a syscall is made. If in is 2, then we invoke the get_free helper function. In this function we start with the address of kmem.freelist and iterate through it until we reach NULL to signify the end of the list. At the end we return the number of free pages that were found which is then returned by sysinfo. Finally, if in is none of the above values, then we return -1.

procinfo

For procinfo we start with getting the current process' PCB and creating a new pinfo struct. We calculate the number of pages being used by getting the size of the processes memory through the sz element of the PCB. Then then divide that by the PGSIZE (which we accomplish by doing a right shift PGSHIFT number of times), and account for non-whole pages being used by checking p->sz % PGSIZE != 0. If this is true then we increment the page count. With the number of pages being used, we can populate the rest of the pinfo struct with the process ID (from p->pid), the page_usage (using the page count calculated earlier), and syscall_count by using the p->syscalls entry of the PCB. Finally we use the copyout function to copy the "kernel pinfo struct" to the user mode address from the pointer argument.

Kernel to User-Mode

After the syscall returns, we increment the number of syscalls made on a per-process and global basis (in the syscall function in kernel/syscall.c). After that function returns, the value is propagated to the user mode and execution of the user program continues.

Contributions

For this lab, as we formed a group mid-assignment, we each did our own implementation and compared results at the end. Additionally we also both contributed to this report.